

The Worker/Wrapper Transformation

ANDY GILL

Galois, Inc., Beaverton, Oregon, USA

GRAHAM HUTTON

School of Computer Science, University of Nottingham, UK

Abstract

The worker/wrapper transformation is a technique for changing the type of a computation, usually with the aim of improving its performance. It has been used by compiler writers for many years, but the technique is little-known in the wider functional programming community, and has never been formalised. In this article we explain, formalise, and explore the generality of the worker/wrapper transformation. We also provide a systematic recipe for its use, and illustrate the power of this recipe using a range of examples.

1 Introduction

The worker/wrapper transformation is a technique for transforming a computation of one type into a *worker* of a different type, together with a *wrapper* that acts as an impedance matcher between the original and new computations.

The technique can be used to improve the performance of functional programs by improving the choice of data structures used. For example, in the Glasgow Haskell Compiler, a specific instance of the worker/wrapper transformation is used to replace boxed data structures by unboxed structures, based upon strictness information (Peyton Jones & Launchbury, 1991). Despite its practical importance, however, the technique is little-known in the wider functional programming community, and to the best of our knowledge has never been formalised.

In this article we explain, formalise, and explore the generality of the worker/wrapper transformation. We also provide a systematic recipe for its use, and illustrate the power of this recipe using a range of programming examples. More precisely, the article makes the following contributions:

- We present the worker/wrapper transformation in a general framework, rather than via specific instances, to allow others to reuse it in their own work.
- We give a correctness proof for the technique, based upon the use of the rolling rule from fixed point calculus. The proof provides a range of explicit conditions under which the technique can be applied.
- We provide a systematic recipe for its use, based upon a single design decision at the start of the transformation, from which point applying the technique is then largely a matter of routine equational reasoning.

- We illustrate this recipe by applying it to four different approaches to improving the performance of programs, based upon the use of accumulation, unboxed structures, memoisation, and continuations.

A summary of related work, and directions for further work, are provided in the concluding sections. The article is aimed at a reader who is familiar with the basics of reasoning about functional programs, say to the level of (Bird, 1998), but no previous experience with the worker/wrapper transformation is assumed. The techniques are presented using Haskell (Peyton Jones, 2003), but can easily be adapted to other functional languages. An extended version of the article that includes all the proofs is available from the authors' web pages.

2 The basic idea

In this section we review the basic idea of the worker/wrapper transformation, in a similar manner to which it was originally described by Peyton Jones and Launchbury (1991), and implemented in the Glasgow Haskell Compiler.

Suppose that we are given a function f , defined in the form

$$f = \textit{body}$$

where \textit{body} is the right-hand side of the definition, and may include free occurrences of f if the function is recursive. The first step in applying the worker/wrapper transformation is to define appropriate functions \textit{wrap} and \textit{unwrap} that allow the function f to be redefined by the equation $f = \textit{wrap} (\textit{unwrap} \textit{body})$, which is then split into two equations by naming the intermediate result:

$$\begin{aligned} f &= \textit{wrap} \textit{work} \\ \textit{work} &= \textit{unwrap} \textit{body} \end{aligned}$$

In this manner, f has been factorised into the application of a “wrapper” function \textit{wrap} to a “worker” function \textit{work} , itself defined by applying \textit{unwrap} to the body of the original definition for f . Note that if f was originally recursive, then f and \textit{work} are now mutually recursive. The next step in the process is to eliminate such mutual recursion by inlining the new definition for f in the definition for \textit{work} , thereby making \textit{work} into a recursive definition that is independent of f :

$$\begin{aligned} f &= \textit{wrap} \textit{work} \\ \textit{work} &= \textit{unwrap} (\textit{body} [\textit{wrap} \textit{work} / f]) \end{aligned}$$

As usual, $e[e'/x]$ denotes the result of substituting e' for all free occurrences of the variable x in the expression e . The final step is then to simplify the resulting definitions for f and \textit{work} using standard techniques.

For example, Peyton Jones and Launchbury (1991) show how this process can be used to transform a recursive definition for the factorial function into a more efficient version that is defined in terms of a worker that only uses unboxed integers, together with a wrapper that takes care of the initial unboxing and final boxing.

The above, syntactic, description of the worker/wrapper transformation is appealingly simple, but raises a number of important questions. Is the technique actually correct? How can this be proved? Under what conditions does it hold? How should it be used in practice? What kind of applications is it suitable for? Addressing these questions is the purpose of the remainder of this article.

3 The worker/wrapper transformation

The key to formalising the worker/wrapper transformation, and hence proving its correctness, is being explicit about the fixed point semantics of recursive definitions, for which purposes we define a fixed point operator in Haskell as follows:

$$\begin{aligned} \text{fix} &:: (a \rightarrow a) \rightarrow a \\ \text{fix } f &= f (\text{fix } f) \end{aligned}$$

The property of this operator that we will use to formalise the worker/wrapper transformation is the *rolling rule* (Backhouse, 2002), which allows us to pull the first argument of a composition outside a fixed point, resulting in the composition swapping the order of its arguments, or “rolling over”:

$$\text{fix } (g \circ f) = g (\text{fix } (f \circ g))$$

Informally, this rule is valid because both sides expand to the infinite application $g (f (g (f (g (f \dots))))))$. A formal proof, together with a brief review of the fixed point approach to recursion, is provided in the appendix.

Now consider the problem of changing the type of a recursive computation. More precisely, suppose that we are given a computation $\text{comp} :: A$, defined as the least fixed point $\text{comp} = \text{fix } \text{body}$ of some function $\text{body} :: A \rightarrow A$, and that we wish to change the underlying type from A to some other type B . For example, it may be that B supports a more efficient implementation of the computation.

The worker/wrapper approach to this problem is based upon defining conversion functions $\text{unwrap} :: A \rightarrow B$ and $\text{wrap} :: B \rightarrow A$ between the two types, such that the equation $\text{wrap} \circ \text{unwrap} = \text{id}_A$ holds, where id_A is the identity function for the type A . This equation states that converting a value of the original type into the new type and then back again does not change the value, and formalises the idea that the type A can be faithfully represented by the type B .

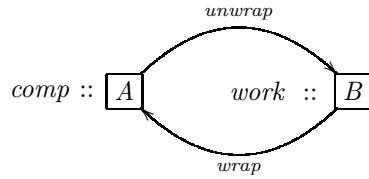
Given such a setting, we can now calculate as follows:

$$\begin{aligned} &\text{comp} \\ = &\quad \{ \text{applying } \text{comp} \} \\ &\text{fix } \text{body} \\ = &\quad \{ \text{id is the identity for } \circ \} \\ &\text{fix } (\text{id}_A \circ \text{body}) \\ = &\quad \{ \dagger \text{ assuming } \text{wrap} \circ \text{unwrap} = \text{id}_A \} \\ &\text{fix } (\text{wrap} \circ \text{unwrap} \circ \text{body}) \\ = &\quad \{ \text{rolling rule} \} \\ &\text{wrap } (\text{fix } (\text{unwrap} \circ \text{body} \circ \text{wrap})) \end{aligned}$$

$$= \quad \{ \text{define } work = fix (unwrap \circ body \circ wrap) \}$$

$$wrap \ work$$

That is, we have shown how an arbitrary recursive computation of type A can be factorised as a *wrapper* of type $B \rightarrow A$ applied to a new recursive *worker* computation of type B , based upon the assumption that the identity function for A can be split into the composition of two conversion functions. The following diagram summarises the primary typing relationships:



For some applications, however, the assumption $wrap \circ unwrap = id_A$ may not be true in general, but only in the context (†) in which it is used in the above calculation. For example, we may sometimes require the weaker property $wrap \circ unwrap \circ body = id_A \circ body$, which states that the assumption is true in the context of values produced by *body*, or even $fix (wrap \circ unwrap \circ body) = fix (id_A \circ body)$, which also takes the recursive context into account. That is, we have the following hierarchy of *worker/wrapper assumptions* that support the above calculation:

$$wrap \circ unwrap = id_A$$

$$\Downarrow$$

$$wrap \circ unwrap \circ body = body$$

$$\Downarrow$$

$$fix (wrap \circ unwrap \circ body) = fix \ body$$

For any given application, we will use the strongest such assumption that is valid, which in practice is often just the basic equation $wrap \circ unwrap = id_A$. In conclusion, the original syntactic description of the worker/wrapper transformation can now be formalised on semantic grounds as follows:

If $comp :: A$ is defined by $comp = fix \ body$ for some $body :: A \rightarrow A$, and $wrap :: B \rightarrow A$ and $unwrap :: A \rightarrow B$ satisfy any of the worker/wrapper assumptions, then

$$comp = wrap \ work$$

where $work :: B$ is defined by

$$work = fix (unwrap \circ body \circ wrap)$$

Once this transformation has been applied, one then attempts to simplify the

definition of the worker using normal equational reasoning techniques. In most cases, the driving force for this process will be the desire to fuse together instances of *unwrap* and *wrap*, in order to eliminate the overhead of repeatedly converting between the new and original types. In general, it is not the case that $unwrap \circ wrap = id_B$, but the following *worker/wrapper fusion* property, in which the argument value of type B is the worker itself, is often applicable:

$$\text{If } wrap \circ unwrap = id_A, \text{ then } (unwrap \circ wrap) \text{ work} = \text{work}$$

This fusion property can be verified as follows:

$$\begin{aligned}
& (unwrap \circ wrap) \text{ work} \\
= & \quad \{ \text{applying } work \} \\
& (unwrap \circ wrap) (\text{fix } (unwrap \circ body \circ wrap)) \\
= & \quad \{ id \text{ is the identity for } \circ \} \\
& (unwrap \circ wrap) (\text{fix } (unwrap \circ body \circ id_A \circ wrap)) \\
= & \quad \{ \text{assuming } wrap \circ unwrap = id_A \} \\
& (unwrap \circ wrap) (\text{fix } (unwrap \circ body \circ wrap \circ unwrap \circ wrap)) \\
= & \quad \{ \text{rolling rule} \} \\
& \text{fix } (unwrap \circ wrap \circ unwrap \circ body \circ wrap) \\
= & \quad \{ \text{assuming } wrap \circ unwrap = id_A \} \\
& \text{fix } (unwrap \circ id_A \circ body \circ wrap) \\
= & \quad \{ id \text{ is the identity for } \circ \} \\
& \text{fix } (unwrap \circ body \circ wrap) \\
= & \quad \{ \text{unapplying } work \} \\
& \text{work}
\end{aligned}$$

In summary, we have the following general recipe for applying the worker/wrapper transformation to change the type of a recursive computation:

- Express the computation as a least fixed point;
- Choose the desired new type for the computation;
- Define conversions between the original and new types;
- Check they satisfy one of the worker/wrapper assumptions;
- Apply the worker/wrapper transformation;
- Simplify the resulting definitions.

We conclude this section by noting that the worker/wrapper transformation can also be formalised using *fixed point fusion* (Meijer *et al.*, 1991) which requires the additional property that *wrap* is strict ($wrap \perp = \perp$). However, this property follows automatically from the basic worker/wrapper assumption $wrap \circ unwrap = id$, by virtue of the fact that any (monotonic) function that is right invertible must be strict. We prefer the above formulation using the rolling rule because it corresponds directly to our intuition for why the transformation is valid.

4 Example: the reverse function

As a first example of the use of our worker/wrapper recipe, we will show how it can be used to transform a simple definition for the function that reverses a list into a more efficient version that uses the technique of *accumulation*.

4.1 Hughes lists

We begin by reviewing the idea of representing lists using functions, which is the essential algorithmic idea underlying this example.

As observed by Hughes (1986), it is possible to represent the concrete type of lists in an alternative manner using functions, by representing a list xs as the function $(xs \#)$ that appends this list to another list that has still to be supplied. We can implement this idea by defining a type $H\ a$ of *Hughes lists*, together with a function $c2a$ that converts concrete (normal) lists into alternative (Hughes) lists:

$$\begin{aligned} \text{type } H\ a &= [a] \rightarrow [a] \\ c2a &:: [a] \rightarrow H\ a \\ c2a\ xs &= (xs \#) \end{aligned}$$

An important property of this representation is that the function $c2a$ forms a homomorphism from lists to functions, in the sense that:

$$\begin{aligned} c2a\ (xs \# ys) &= c2a\ xs \circ c2a\ ys \\ c2a\ [] &= id \end{aligned}$$

That is, $c2a$ is a homomorphism from the monoid of (finite) lists, for which the associative operator is $\#$ and the unit is $[]$, to the monoid of functions, for which the operator is composition \circ and the unit is the identity function id .

In addition to the function $c2a$ that converts from concrete lists to alternative lists, it is also natural to consider conversion in the opposite direction, which is achieved by simply applying the given function to the empty list:

$$\begin{aligned} a2c &:: H\ a \rightarrow [a] \\ a2c\ f &= f\ [] \end{aligned}$$

This definition ensures that converting a list into Hughes form and then back again gives the original list, or more formally, that $a2c \circ c2a = id$.

4.2 Reverse

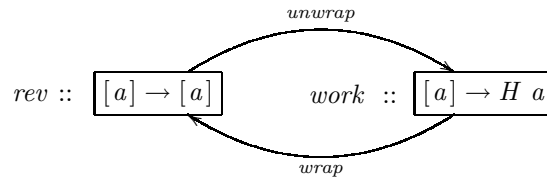
Consider the following simple definition for the function that reverses a list, as found in most introductory functional programming courses:

$$\begin{aligned} rev &:: [a] \rightarrow [a] \\ rev\ [] &= [] \\ rev\ (x : xs) &= rev\ xs \# [x] \end{aligned}$$

Because of the use of $++$, this definition takes quadratic time in the length of its argument. We now show how this definition can be transformed into a more efficient *worker* that uses an extra argument to accumulate the result, together with a *wrapper* that takes care of the initial setup. The first step is to redefine *rev* as a least fixed point, by abstracting over the recursive call in the body:

$$\begin{aligned}
rev &:: [a] \rightarrow [a] \\
rev &= \text{fix } body \\
body &:: ([a] \rightarrow [a]) \rightarrow ([a] \rightarrow [a]) \\
body \ r \ [] &= [] \\
body \ r \ (x : xs) &= r \ xs ++ [x]
\end{aligned}$$

Following our worker/wrapper recipe, and utilising the idea of Hughes lists, our aim now is to transform the computation *rev* of type $[a] \rightarrow [a]$ into a worker of type $[a] \rightarrow H \ a$ (which abbreviates $[a] \rightarrow [a] \rightarrow [a]$ and hence introduces the necessary extra argument list), as illustrated by the following diagram:



Implementing the conversion functions *unwrap* and *wrap* is simply a matter of applying the conversion functions *c2a* and *a2c* from the previous section:

$$\begin{aligned}
unwrap &:: ([a] \rightarrow [a]) \rightarrow ([a] \rightarrow H \ a) \\
unwrap \ f \ xs &= c2a \ (f \ xs) \\
wrap &:: ([a] \rightarrow H \ a) \rightarrow ([a] \rightarrow [a]) \\
wrap \ f \ xs &= a2c \ (f \ xs)
\end{aligned}$$

Verifying the worker/wrapper assumption, in this case $wrap \circ unwrap = id$, follows from the fact that $a2c \circ c2a = id$ by straightforward calculation.

Now that we have satisfied the preconditions for the worker/wrapper transformation, applying this transformation gives the following result:

$$\begin{aligned}
rev &:: [a] \rightarrow [a] \\
rev &= wrap \ work \\
work &:: [a] \rightarrow H \ a \\
work &= \text{fix } (unwrap \circ body \circ wrap)
\end{aligned}$$

Making the list argument explicit in the definition for *rev*, and expanding out *wrap*, we obtain the expected wrapper for the efficient version of *rev*, which simply

supplies the empty list as the initial accumulator:

$$\begin{aligned} rev &:: [a] \rightarrow [a] \\ rev\ xs &= work\ xs\ [] \end{aligned}$$

Now let us focus our attention on simplifying the definition for the worker function. First of all, we redefine *work* using explicit recursion

$$\begin{aligned} work\ [] &= c2a\ [] \\ work\ (x : xs) &= c2a\ (wrap\ work\ xs \# [x]) \end{aligned}$$

which transformation can be verified as follows:

$$\begin{aligned} &work\ xs \\ = &\quad \{ \text{applying } work \} \\ &fix\ (unwrap \circ body \circ wrap)\ xs \\ = &\quad \{ \text{applying } fix \} \\ &(unwrap \circ body \circ wrap)\ (fix\ (unwrap \circ body \circ wrap))\ xs \\ = &\quad \{ \text{unapplying } work \} \\ &(unwrap \circ body \circ wrap)\ work\ xs \\ = &\quad \{ \text{applying } \circ \} \\ &unwrap\ (body\ (wrap\ work))\ xs \\ = &\quad \{ \text{applying } unwrap \} \\ &c2a\ (body\ (wrap\ work)\ xs) \\ = &\quad \{ \text{applying } body \} \\ &c2a\ (\mathbf{case}\ xs\ \mathbf{of} \\ &\quad [] \rightarrow [] \\ &\quad (x : xs) \rightarrow wrap\ work\ xs \# [x]) \\ = &\quad \{ \text{distribution over } \mathbf{case} \} \\ &\mathbf{case}\ xs\ \mathbf{of} \\ &\quad [] \rightarrow c2a\ [] \\ &\quad (x : xs) \rightarrow c2a\ (wrap\ work\ xs \# [x]) \end{aligned}$$

The distribution step relies on the fact that *c2a* is strict ($c2a\ \perp = \perp$), which reduces to showing that $(\lambda xs \rightarrow \perp) = \perp$. This equation holds in Haskell, provided that we prohibit the use of language features that are used to force evaluation of expressions, such as *seq* (Peyton Jones, 2003). Alternatively, the distribution step is also valid if we assume that lists are never undefined ($= \perp$).

Now we exploit the fact that $c2a :: [a] \rightarrow H\ a$ is a homomorphism from lists to functions, to rewrite the new definition as follows:

$$\begin{aligned} work\ [] &= id \\ work\ (x : xs) &= c2a\ (wrap\ work\ xs) \circ c2a\ [x] \end{aligned}$$

Then we simplify the first argument of the composition in this definition by exposing the implicit use of *unwrap* and then fusing this with *wrap*:

$$\begin{aligned} &c2a\ (wrap\ work\ xs) \\ = &\quad \{ \text{unapplying } unwrap \} \end{aligned}$$

Such tagging supports the use of non-strict evaluation, but carries a considerable overhead. For example, evaluating $x + y$ conceptually requires evaluating the expressions x and y , unboxing the resulting values (removing the tags), performing the actual addition, and then boxing the result (reinstating the tag). It would clearly be more efficient to work with unboxed integers as much as possible.

The key to achieving this behaviour in Haskell is an idea put forward by Peyton Jones and Launchbury (1991), namely to reflect the notion of boxed and unboxed integers directly in the language itself, via the following type definition:

$$\mathbf{data} \text{ Int} = I_{\#} \text{ Int}_{\#}$$

That is, a boxed value of type Int can be formed by applying the constructor tag $I_{\#}$ to a value of type $\text{Int}_{\#}$. In turn, $\text{Int}_{\#}$ is the built-in type of unboxed 32-bit integers, which is unlifted in the sense that it has no \perp element. Note that, by convention in Haskell, unboxed types, values, variables and functions have a $\#$ suffix on their name, as in $\text{Int}_{\#}$, $0_{\#}$, $n_{\#}$, and $+_{\#}$. However, this symbol has no semantic meaning, and is used purely for identification purposes.

Given the above type definition, converting an unboxed integer into a boxed integer can be achieved simply by applying $I_{\#}$, and conversion in the opposite direction by removing this constructor using pattern matching, so there is no need to define explicit conversion functions between the two types.

5.2 Factorial

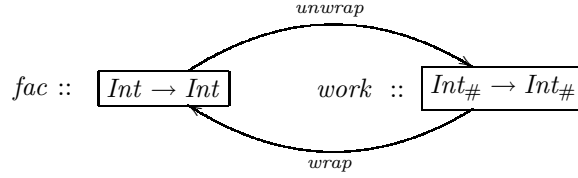
Consider the following definition for the factorial function on non-negative integers, which is often used to illustrate the idea of recursion:

$$\begin{aligned} \text{fac} &:: \text{Int} \rightarrow \text{Int} \\ \text{fac } 0 &= 1 \\ \text{fac } n &= n * \text{fac } (n - 1) \end{aligned}$$

In a non-strict language such as Haskell, this definition may be rather inefficient, due to repeated unboxing and boxing of integer values. However, because fac is strict ($\text{fac } \perp = \perp$), it is safe to replace the definition of fac by a more efficient *worker* that only uses unboxed integers and strict evaluation, together with a *wrapper* that takes care of the initial unboxing and final reboxing. We now show how this transformation can be achieved in practice. As previously, the first step in applying our worker/wrapper recipe is to redefine fac as a least fixed point:

$$\begin{aligned} \text{fac} &:: \text{Int} \rightarrow \text{Int} \\ \text{fac} &= \text{fix } \text{body} \\ \text{body} &:: (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int}) \\ \text{body } f \ 0 &= 1 \\ \text{body } f \ n &= n * f \ (n - 1) \end{aligned}$$

Our aim now is to transform the computation fac of type $\text{Int} \rightarrow \text{Int}$ into a worker of type $\text{Int}_{\#} \rightarrow \text{Int}_{\#}$, as illustrated in the following diagram:



Implementing the conversion functions *wrap* and *unwrap* is simply a matter of taking care of the necessary boxing and unboxing:

$$\begin{aligned}
 \text{unwrap} &:: (Int \rightarrow Int) \rightarrow (Int_{\#} \rightarrow Int_{\#}) \\
 \text{unwrap } f \ x_{\#} &= \mathbf{case} \ f \ (I_{\#} \ x_{\#}) \ \mathbf{of} \\
 &\quad I_{\#} \ y_{\#} \rightarrow y_{\#} \\
 \text{wrap} &:: (Int_{\#} \rightarrow Int_{\#}) \rightarrow (Int \rightarrow Int) \\
 \text{wrap } f_{\#} \ x &= \mathbf{case} \ x \ \mathbf{of} \\
 &\quad I_{\#} \ x_{\#} \rightarrow \mathbf{case} \ f_{\#} \ x_{\#} \ \mathbf{of} \\
 &\quad \quad y_{\#} \rightarrow I_{\#} \ y_{\#}
 \end{aligned}$$

Using the above definitions and properties of case expressions, it is straightforward to show that $\text{wrap} \circ \text{unwrap} = id$, but only if we restrict our attention to argument functions of type $Int \rightarrow Int$ that are strict. Hence for this example, the simple worker/wrapper assumption $\text{wrap} \circ \text{unwrap} = id$ is not always valid. However, because *body* *f* is defined by pattern matching and is therefore strict, we do have the weaker assumption $\text{wrap} \circ \text{unwrap} \circ \text{body} = \text{body}$, which is precisely where strictness is exploited for this example. More complex functions than factorial whose strictness depends upon taking account of recursion will require our weakest worker/wrapper assumption, defined using *fix*.

Now that we have satisfied the worker/wrapper preconditions, applying this transformation gives the following result:

$$\begin{aligned}
 \text{fac} &:: Int \rightarrow Int \\
 \text{fac} &= \text{wrap } \text{work} \\
 \text{work} &:: Int_{\#} \rightarrow Int_{\#} \\
 \text{work} &= \text{fix } (\text{unwrap} \circ \text{body} \circ \text{wrap})
 \end{aligned}$$

Making the integer argument explicit in the definition for *fac*, and expanding out *wrap*, we obtain the expected wrapper for the unboxed version of *fac*, which unboxes the integer argument, applies the worker, and then boxes the result:

$$\begin{aligned}
 \text{fac} &:: Int \rightarrow Int \\
 \text{fac } n &= \mathbf{case} \ n \ \mathbf{of} \\
 &\quad I_{\#} \ x_{\#} \rightarrow \mathbf{case} \ \text{work } x_{\#} \ \mathbf{of} \\
 &\quad \quad y_{\#} \rightarrow I_{\#} \ y_{\#}
 \end{aligned}$$

Now let us simplify the definition for the worker function, which in this case amounts to expanding out definitions and then simplifying the result. As with the *reverse* example, the first step is to redefine *work* using explicit recursion, which

transformation can be verified in a similar manner to previously:

$$\begin{aligned}
\text{work } x_{\#} &= \mathbf{case} (\mathbf{case} (I_{\#} x_{\#}) \mathbf{of} \\
&\quad 0 \rightarrow 1 \\
&\quad n \rightarrow \mathbf{case} (n - 1) \mathbf{of} \\
&\quad\quad I_{\#} a_{\#} \rightarrow \mathbf{case} \text{work } a_{\#} \mathbf{of} \\
&\quad\quad\quad b_{\#} \rightarrow n * (I_{\#} b_{\#})) \mathbf{of} \\
&\quad I_{\#} y_{\#} \rightarrow y_{\#}
\end{aligned}$$

Now we expand out the boxed values 0 and 1, together with the boxed variable n , in terms of the constructor $I_{\#}$, to rewrite the new definition as follows:

$$\begin{aligned}
\text{work } x_{\#} &= \mathbf{case} (\mathbf{case} (I_{\#} x_{\#}) \mathbf{of} \\
&\quad I_{\#} 0_{\#} \rightarrow I_{\#} 1_{\#} \\
&\quad I_{\#} n_{\#} \rightarrow \mathbf{case} ((I_{\#} n_{\#}) - (I_{\#} 1_{\#})) \mathbf{of} \\
&\quad\quad I_{\#} a_{\#} \rightarrow \mathbf{case} \text{work } a_{\#} \mathbf{of} \\
&\quad\quad\quad b_{\#} \rightarrow (I_{\#} n_{\#}) * (I_{\#} b_{\#})) \mathbf{of} \\
&\quad I_{\#} y_{\#} \rightarrow y_{\#}
\end{aligned}$$

Then we expand out the subtraction $-$ and multiplication $*$ operators on boxed integers, using the fact that for any such operator \oplus we have

$$(I_{\#} x_{\#}) \oplus (I_{\#} y_{\#}) = \mathbf{case} (x_{\#} \oplus_{\#} y_{\#}) \mathbf{of} \\
z_{\#} \rightarrow I_{\#} z_{\#}$$

to give the following result:

$$\begin{aligned}
\text{work } x_{\#} &= \mathbf{case} (\mathbf{case} (I_{\#} x_{\#}) \mathbf{of} \\
&\quad I_{\#} 0_{\#} \rightarrow I_{\#} 1_{\#} \\
&\quad I_{\#} n_{\#} \rightarrow \mathbf{case} (\mathbf{case} n_{\#} \text{ }_{\#} 1_{\#} \mathbf{of} \\
&\quad\quad c_{\#} \rightarrow I_{\#} c_{\#}) \mathbf{of} \\
&\quad\quad I_{\#} a_{\#} \rightarrow \mathbf{case} \text{work } a_{\#} \mathbf{of} \\
&\quad\quad\quad b_{\#} \rightarrow \mathbf{case} n_{\#} *_{\#} b_{\#} \mathbf{of} \\
&\quad\quad\quad d_{\#} \rightarrow I_{\#} d_{\#}) \mathbf{of} \\
&\quad I_{\#} y_{\#} \rightarrow y_{\#}
\end{aligned}$$

Finally, if we simplify this definition using properties of case, we obtain the following definition for the worker, which operates entirely using unboxed integers, and hence avoids the need for repeated unboxing and boxing:

$$\begin{aligned}
\text{work} &:: \text{Int}_{\#} \rightarrow \text{Int}_{\#} \\
\text{work } n_{\#} &= \mathbf{case} n_{\#} \mathbf{of} \\
&\quad 0_{\#} \rightarrow 1_{\#} \\
&\quad n_{\#} \rightarrow \mathbf{case} n_{\#} \text{ }_{\#} 1_{\#} \mathbf{of} \\
&\quad\quad a_{\#} \rightarrow \mathbf{case} \text{work } a_{\#} \mathbf{of} \\
&\quad\quad\quad b_{\#} \rightarrow n_{\#} *_{\#} b_{\#}
\end{aligned}$$

Once again, note that once we have made the initial decision regarding the change in type of the function, applying our worker/wrapper receipt is largely a matter of routine equational reasoning, and does not require the use of induction.

6 Example: the Fibonacci function

For our next example, we show how the worker/wrapper transformation can be used to transform a simple definition for the Fibonacci function on natural numbers into a more efficient version that operates by means of a *memo table*.

6.1 Memoisation

The term memoisation was coined by Michie (1968), and refers to the optimisation technique of tabulating all the arguments that a function is called with, together with the corresponding results returned, and reusing these results if the function is called again with any of these arguments. If this happens often, memoisation can lead to a dramatic improvement in performance.

Our approach to memoising the Fibonacci function is to observe that any function on natural numbers can be represented in an alternative manner as an infinite list (or stream) of results. In particular, we represent a function f as the stream $[f\ 0, f\ 1, f\ 2, \dots]$ that tabulates its behaviour for all possible argument values.

To implement this representation, let us write Nat for the subtype of Int comprising the natural numbers $0, 1, 2, 3, \dots$, and $Stream\ a$ for the subtype of $[a]$ comprising the infinite lists of elements of type a . Using these two subtypes allows us to be more precise about the types of functions that we will define, while still being able to utilise familiar notation and library functions for integers and lists.

For example, a function $unwrap$ that converts functions on natural numbers into streams in the manner described above can now be defined as follows:

$$\begin{aligned} unwrap &:: (Nat \rightarrow a) \rightarrow Stream\ a \\ unwrap\ f &= map\ f\ [0..] \end{aligned}$$

For the purposes of proofs, however, an equivalent definition that uses explicit co-recursion (Gibbons & Hutton, 2005) is usually preferable:

$$unwrap\ f = f\ 0 : unwrap\ (f \circ (+1))$$

Dually, we can also perform conversion in the opposite direction, from a stream to a function, by simply indexing into the stream:

$$\begin{aligned} wrap &:: Stream\ a \rightarrow (Nat \rightarrow a) \\ wrap\ xs &= (xs\ !!) \end{aligned}$$

The auxiliary operator $!!$ selects the n th element of a stream:

$$\begin{aligned} (!! &:: Stream\ a \rightarrow Nat \rightarrow a \\ xs\ !!\ 0 &= head\ xs \\ xs\ !!\ (n + 1) &= (tail\ xs)\ !!\ n \end{aligned}$$

Using these definitions, we can show that $wrap \circ unwrap = id$ by first making the function and numeric arguments explicit, then expanding out definitions to give $(unwrap\ f)\ !!\ n = f\ n$, and finally verifying this equation by induction on n .

The dual property, $unwrap \circ wrap = id$, also holds. In particular, making the

stream argument explicit and expanding out definitions gives $unwrap (xs !!) = xs$, which can then be verified by coinduction on streams (Turner, 1995).

In conclusion, the functions $unwrap$ and $wrap$ establish an isomorphism between the types $Nat \rightarrow a$ and $Stream\ a$. This isomorphism itself is well-known, but perhaps surprisingly, generalises to allow any function type with an inductively-defined argument type to be represented in a coinductive manner (Altenkirch, 2001), thereby providing a basis for memoising a large class of functions.

6.2 Fibonacci

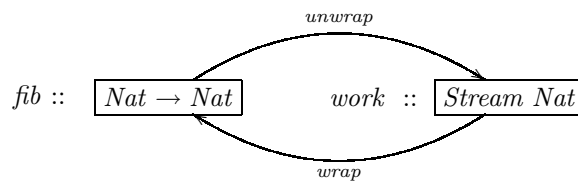
Consider the following standard definition for the Fibonacci function on natural numbers, as found in any textbook on numerical series:

$$\begin{aligned} fib &:: Nat \rightarrow Nat \\ fib\ 0 &= 1 \\ fib\ 1 &= 1 \\ fib\ (n + 2) &= fib\ n + fib\ (n + 1) \end{aligned}$$

That is, the first two Fibonacci numbers are 1, and each successive number is the sum of the previous two. While this definition directly expresses the desired behaviour of the function, it also recomputes the same results many times over, in fact requiring exponential time in the size of its argument. We now show how it can be transformed into a more efficient *worker* that constructs a memo table comprising the stream $[1, 1, 2, 3, 5, 8, \dots]$ of all Fibonacci numbers (and hence ensures that each result is computed at most once), together with a *wrapper* that selects the required element from this table. The first step is to redefine fib using fix :

$$\begin{aligned} fib &:: Nat \rightarrow Nat \\ fib &= fix\ body \\ body &:: (Nat \rightarrow Nat) \rightarrow (Nat \rightarrow Nat) \\ body\ f\ 0 &= 1 \\ body\ f\ 1 &= 1 \\ body\ f\ (n + 2) &= f\ n + f\ (n + 1) \end{aligned}$$

Our aim now is to transform the computation fib of type $Nat \rightarrow Nat$ into a worker of type $Stream\ Nat$, as illustrated in the following diagram:



Because we have already defined appropriate conversion functions satisfying the assumption $unwrap \circ wrap = id$ in the previous section, we can now apply the

worker/wrapper transformation to obtain the following result:

$$\begin{aligned}
 \mathit{fib} &:: \mathit{Nat} \rightarrow \mathit{Nat} \\
 \mathit{fib} &= \mathit{wrap} \ \mathit{work} \\
 \mathit{work} &:: \mathit{Stream} \ \mathit{Nat} \\
 \mathit{work} &= \mathit{fix} \ (\mathit{unwrap} \circ \mathit{body} \circ \mathit{wrap})
 \end{aligned}$$

Making the numeric argument explicit in the definition for fib , and expanding out wrap , we obtain the expected wrapper for the memoised version of fib , which simply selects the required element from the memo table:

$$\begin{aligned}
 \mathit{fib} &:: \mathit{Nat} \rightarrow \mathit{Nat} \\
 \mathit{fib} \ n &= \mathit{work} \ !! \ n
 \end{aligned}$$

Now we simplify the worker itself, which in this case just amounts to expanding out definitions, and does not require any form of fusion:

$$\begin{aligned}
 &\mathit{work} \\
 = &\quad \{ \text{applying } \mathit{work} \} \\
 &\mathit{fix} \ (\mathit{unwrap} \circ \mathit{body} \circ \mathit{wrap}) \\
 = &\quad \{ \text{applying } \mathit{fix} \} \\
 &(\mathit{unwrap} \circ \mathit{body} \circ \mathit{wrap}) \ (\mathit{fix} \ (\mathit{unwrap} \circ \mathit{body} \circ \mathit{wrap})) \\
 = &\quad \{ \text{unapplying } \mathit{work} \} \\
 &(\mathit{unwrap} \circ \mathit{body} \circ \mathit{wrap}) \ \mathit{work} \\
 = &\quad \{ \text{applying } \circ \} \\
 &\mathit{unwrap} \ (\mathit{body} \ (\mathit{wrap} \ \mathit{work})) \\
 = &\quad \{ \text{applying } \mathit{unwrap} \} \\
 &\mathit{map} \ (\mathit{body} \ (\mathit{wrap} \ \mathit{work})) \ [0..] \\
 = &\quad \{ \text{applying } \mathit{body} \} \\
 &\mathit{map} \ (\lambda n \rightarrow \mathbf{case} \ n \ \mathbf{of} \\
 &\quad 0 \rightarrow 1 \\
 &\quad 1 \rightarrow 1 \\
 &\quad (n + 2) \rightarrow \mathit{wrap} \ \mathit{work} \ n + \mathit{wrap} \ \mathit{work} \ (n + 1)) \ [0..] \\
 = &\quad \{ \text{applying } \mathit{wrap} \} \\
 &\mathit{map} \ (\lambda n \rightarrow \mathbf{case} \ n \ \mathbf{of} \\
 &\quad 0 \rightarrow 1 \\
 &\quad 1 \rightarrow 1 \\
 &\quad (n + 2) \rightarrow \mathit{work} \ !! \ n + \mathit{work} \ !! \ (n + 1)) \ [0..]
 \end{aligned}$$

That is, we have derived the following definition for the worker, which builds a memo table comprising all Fibonacci numbers, using the table itself to ensure that each number is computed at most once, and lazy evaluation to ensure that the table

(which is conceptually infinite) is built on a demand-driven basis:

```

work  :: Stream Nat
work  = map f [0..]
where
    f 0 = 1
    f 1 = 1
    f (n + 2) = work !! n + work !! (n + 1)

```

In conclusion, we have improved the time performance of the *fib* function from exponential to quadratic. Further improvements can readily be made in a variety of different ways, such as by representing the memo table in a more efficient manner (e.g. using an array with constant-time indexing rather than a stream with linear-time indexing), by exploiting the structure of the computation to design a specialised version of the memo table (e.g. using a pair of numbers rather than a stream), or by using more advanced programming techniques (e.g. cyclic programming). We prefer the stream version here because it is more general, in the sense that streams are the natural memo structure for any function on (Peano) natural numbers, and do not rely on intensional properties of the function itself.

7 Example: an evaluation function

For our final example, we will show how the worker/wrapper transformation can be used to implement the process of transforming a function into *continuation-passing* style, an important precursor to many program analyses and optimisations.

7.1 Continuations

As observed by Reynolds (1972), it is possible to represent any type in an alternative manner using *continuations*. The notion of a continuation can be defined in many ways, but for our purposes is simply a function that is applied to the result of another computation. Using this idea, we represent a value x as the function $\lambda c \rightarrow c x$ that takes another function c (a continuation) as its argument, and applies this function to x in order to produce the final result.

We can implement this representation by defining a type *Cont a* of continuation computations of type a , together with a function *c2a* that converts concrete (normal) values into alternative (continuation) values:

```

type Cont a = (a → a) → a
c2a      :: a → Cont a
c2a x    = λc → c x

```

The continuation type can be generalised to $(a \rightarrow b) \rightarrow b$, but we do not need the extra generality here. Dually, we can also perform conversion in the opposite

direction, by supplying the identity function as the continuation:

$$\begin{aligned} a2c &:: \text{Cont } a \rightarrow a \\ a2c f &= f \text{ id} \end{aligned}$$

Using these definitions, it is easy to show that $a2c \circ c2a = \text{id}$.

7.2 Evaluation

Consider a simple expression language comprising integers values, an addition operator, a single exceptional value called throw, and a catch operator:

$$\mathbf{data} \text{ Expr} = \text{Val Int} \mid \text{Add Expr Expr} \mid \text{Throw} \mid \text{Catch Expr Expr}$$

This language provides an appropriate minimal setting in which to investigate various aspects of the semantics of exceptions (Hutton & Wright, 2004; 2006; 2007). Informally, *Throw* abandons the current computation and throws an exception, while *Catch x y* behaves as the expression *x* unless it throws an exception, in which case the catch behaves as the handler expression *y*.

To formalise the meaning of this language, we first recall the *Maybe* type:

$$\mathbf{data} \text{ Maybe } a = \text{Nothing} \mid \text{Just } a$$

That is, a value of type *Maybe a* is either *Nothing*, which we think of as an exceptional value, or has the form *Just x* for some *x* of type *a*, which we think of as a normal value (Spivey, 1990). For the purposes of our example, however, we only require the type *Maybe Int*, which we abbreviate by *Mint*:

$$\mathbf{type} \text{ Mint} = \text{Maybe Int}$$

Using this type, it is straightforward to define a function that evaluates expressions, and takes care of the necessary propagation and handling of exceptions:

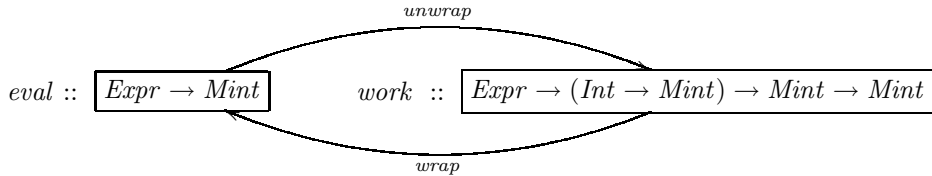
$$\begin{aligned} \text{eval} &:: \text{Expr} \rightarrow \text{Mint} \\ \text{eval} (\text{Val } n) &= \text{Just } n \\ \text{eval} (\text{Add } x \ y) &= \mathbf{case} \ \text{eval } x \ \mathbf{of} \\ &\quad \text{Nothing} \rightarrow \text{Nothing} \\ &\quad \text{Just } n \rightarrow \mathbf{case} \ \text{eval } y \ \mathbf{of} \\ &\quad \quad \text{Nothing} \rightarrow \text{Nothing} \\ &\quad \quad \text{Just } m \rightarrow \text{Just } (n + m) \\ \text{eval} (\text{Throw}) &= \text{Nothing} \\ \text{eval} (\text{Catch } x \ y) &= \mathbf{case} \ \text{eval } x \ \mathbf{of} \\ &\quad \text{Nothing} \rightarrow \text{eval } y \\ &\quad \text{Just } n \rightarrow \text{Just } n \end{aligned}$$

Because of the repeated tagging and untagging of values of type *Mint*, this definition may be rather inefficient. We now show how it can be transformed into a *worker* that uses two continuations to make normal and exceptional control flow explicit (and hence eliminate the use of tags), together with a *wrapper* that takes care of

the initial setup. Once again, the first step is to redefine *eval* using *fix*:

$$\begin{aligned}
eval &:: Expr \rightarrow Mint \\
eval &= fix\ body \\
body &:: (Expr \rightarrow Mint) \rightarrow (Expr \rightarrow Mint) \\
body\ f\ (Val\ n) &= Just\ n \\
body\ f\ (Add\ x\ y) &= \mathbf{case}\ f\ x\ \mathbf{of} \\
&\quad Nothing \rightarrow Nothing \\
&\quad Just\ n \rightarrow \mathbf{case}\ f\ y\ \mathbf{of} \\
&\quad\quad Nothing \rightarrow Nothing \\
&\quad\quad Just\ m \rightarrow Just\ (n + m) \\
body\ f\ (Throw) &= Nothing \\
body\ f\ (Catch\ x\ y) &= \mathbf{case}\ f\ x\ \mathbf{of} \\
&\quad Nothing \rightarrow f\ y \\
&\quad Just\ n \rightarrow Just\ n
\end{aligned}$$

One might expect now to aim to transform the computation *eval* of type $Expr \rightarrow Mint$ into a worker of type $Expr \rightarrow Cont\ Mint$. In practice, however, the desire to have separate continuations for normal and exceptional control flow, sometimes called *double-barrelled* continuations (Thielecke, 2002), means that we start by splitting the underlying type $Mint \rightarrow Mint$ of continuations into two parts, and aim for a worker of the type illustrated in the following diagram:



Intuitively, the second argument of the worker, a continuation of type $Int \rightarrow Mint$, deals with successful evaluation, in which the resulting integer value is then transformed into a value of type $Mint$ by applying this continuation. In turn, the third argument, a value of type $Mint$, which we can view as a continuation with no arguments, handles the case of failure, by directly providing a resulting value of type $Mint$. The necessary conversion functions, satisfying the worker/wrapper assumption $wrap \circ unwrap = id$, can be defined as follows:

$$\begin{aligned}
unwrap\ g\ e\ s\ f &= \mathbf{case}\ (g\ e)\ \mathbf{of} \\
&\quad Nothing \rightarrow f \\
&\quad Just\ n \rightarrow s\ n \\
wrap\ g\ e &= g\ e\ Just\ Nothing
\end{aligned}$$

Now that we have established the preconditions for the worker/wrapper trans-

formation, applying this transformation gives the following result:

$$\begin{aligned}
eval &:: Expr \rightarrow Mint \\
eval &= wrap\ work \\
work &:: Expr \rightarrow (Int \rightarrow Mint) \rightarrow Mint \rightarrow Mint \\
work &= fix\ (unwrap \circ body \circ wrap)
\end{aligned}$$

Making the expression argument explicit in the definition for *eval*, and expanding out *wrap*, we obtain the expected wrapper for the continuation-passing version of *eval*, which simply supplies *Just* and *Nothing* as the initial continuations:

$$\begin{aligned}
eval &:: Expr \rightarrow Mint \\
eval\ e &= work\ e\ Just\ Nothing
\end{aligned}$$

Now let us focus our attention on simplifying the definition for the worker function. First of all, we redefine *work* using explicit recursion:

$$\begin{aligned}
work\ (Val\ n)\ s\ f &= s\ n \\
work\ (Add\ x\ y)\ s\ f &= \mathbf{case}\ wrap\ work\ x\ \mathbf{of} \\
&\quad Nothing \rightarrow f \\
&\quad Just\ n \rightarrow \mathbf{case}\ wrap\ work\ y\ \mathbf{of} \\
&\quad\quad Nothing \rightarrow f \\
&\quad\quad Just\ m \rightarrow s\ (n + m) \\
work\ (Throw)\ s\ f &= f \\
work\ (Catch\ x\ y)\ s\ f &= \mathbf{case}\ wrap\ work\ x\ \mathbf{of} \\
&\quad Nothing \rightarrow \mathbf{case}\ wrap\ work\ y\ \mathbf{of} \\
&\quad\quad Nothing \rightarrow f \\
&\quad\quad Just\ n \rightarrow s\ n \\
&\quad Just\ n \rightarrow s\ n
\end{aligned}$$

Now we simplify the case analyses used in the recursive definition by exposing the implicit use of *unwrap* and then fusing again with *wrap*. In particular, for an arbitrary expression *x*, and continuations *f* and *s* of the appropriate types:

$$\begin{aligned}
&\mathbf{case}\ wrap\ work\ x\ \mathbf{of} \\
&\quad Nothing \rightarrow f \\
&\quad Just\ n \rightarrow s\ n \\
= &\quad \{ \text{unapplying } unwrap \} \\
&\quad unwrap\ (wrap\ work)\ x\ s\ f \\
= &\quad \{ \text{unapplying } \circ \} \\
&\quad (unwrap \circ wrap)\ work\ x\ s\ f \\
= &\quad \{ \text{worker/wrapper fusion} \} \\
&\quad work\ x\ s\ f
\end{aligned}$$

Applying this result three times gives our final definition for the worker, in which control flow is now managed by explicit success and failure continuations, rather

than by repeated tagging and untagging of *Maybe* values:

$$\begin{aligned}
\mathit{work} &:: \mathit{Expr} \rightarrow (\mathit{Int} \rightarrow \mathit{Mint}) \rightarrow \mathit{Mint} \rightarrow \mathit{Mint} \\
\mathit{work} (\mathit{Val} \ n) \ s \ f &= s \ n \\
\mathit{work} (\mathit{Add} \ x \ y) \ s \ f &= \mathit{work} \ x \ (\lambda n \rightarrow \mathit{work} \ y \ (\lambda m \rightarrow s \ (n + m))) \ f \ f \\
\mathit{work} (\mathit{Throw}) \ s \ f &= f \\
\mathit{work} (\mathit{Catch} \ x \ y) \ s \ f &= \mathit{work} \ x \ s \ (\mathit{work} \ y \ s \ f)
\end{aligned}$$

Once again, note that everything follows in a straightforward manner once we have made the initial design decision to represent the type *Mint* using two continuations, and that the derivation does not require the use of induction. We conclude this section by noting that the resulting worker function can now readily be transformed into an abstract machine for evaluating expressions (Ager *et al.*, 2003; Hutton & Wright, 2006), resulting in an efficient machine that operates using two control stacks, one for normal evaluation and the other for handling exceptions.

8 Related work

As discussed in the introductory section, the worker/wrapper transformation was first used to exploit strictness information in the Glasgow Haskell Compiler (Peyton Jones & Launchbury, 1991). However, the correctness of the resulting transformation had never been formally verified. During our work on this article we learned that the authors did in fact sketch a proof using the rolling rule, but this proof was never fully developed or published. The underlying strictness analyser in this compiler has been changed and improved several times since its initial implementation (Peyton Jones & Partain, 1993), but the use of the worker/wrapper technique to exploit this information remains exactly the same.

The worker/wrapper transformation has also been used to improve the shortcut approach to deforestation (Gill *et al.*, 1993), an optimisation technique that attempts to remove intermediate data structures from programs. In particular, the shortcut approach performs fusion when functions that produce and consume lists are written in a specific, idiomatic way, and are completely inlined. In the final chapter of his PhD thesis (1996), Gill describes a scheme for abstracting a function that produces a list in a way that utilised the worker/wrapper transformation to reduce the increase in code size that can result from wholesale inlining of definitions. In particular, Gill considered list producing functions of the form

$$f \ x_1 \ \cdots \ x_n \ = \ \mathit{build} \ (\lambda c \ n \rightarrow \mathit{body})$$

which were then split into a wrapper and a worker

$$\begin{aligned}
f \ x_1 \ \cdots \ x_n &= \mathit{build} \ (\lambda c \ n \rightarrow \mathit{work} \ x_1 \ \cdots \ x_n \ c \ n) \\
\mathit{work} \ x_1 \ \cdots \ x_n \ c \ n &= \mathit{body}
\end{aligned}$$

and finally, the new definition for *f* was inlined in the definition for *work*, thereby communicating the deforestation opportunity. However, no formal justification was given for the correctness of the worker/wrapper technique.

Chitil considerably extended Gill’s work in (2000b; 2000a), by generalising to allow for multiple return lists, as directed by his type-inference based approach to deforestation. In particular, Chitil uses a static argument translation (Santos, 1995) to turn a recursive function into a non-recursive function with local recursion, and splits the new, non-recursive function into a worker and wrapper, with detailed justification. He also explores the recursive instance of the worker/wrapper technique as applied to deforestation, acknowledging that “whereas intuitively the semantic correctness of [this approach] is clear, a formal proof is hard”, and suggests that a proof may be possible using improvement theory (Sands, 1998). However, as we have shown in this article, simpler techniques suffice.

Another application of the worker/wrapper transformation has been in changing the order of function arguments, to aid static analysis. In particular, Launchbury and Sheard (1995) use the technique to illustrate how the standard definition

$$\begin{aligned} \text{map} &:: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\ \text{map } f \text{ } xs &= \mathbf{case } xs \mathbf{ of} \\ &\quad [] \rightarrow [] \\ &\quad (x : xs) \rightarrow f \ x : \text{map } f \ xs \end{aligned}$$

can be translated into an equivalent version that is defined using a worker that takes the two arguments in the opposite order:

$$\begin{aligned} \text{map} &:: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\ \text{map } f \text{ } xs &= \text{work } xs \ f \\ \text{work} &:: [a] \rightarrow (a \rightarrow b) \rightarrow [b] \\ \text{work } xs \ f &= \mathbf{case } xs \mathbf{ of} \\ &\quad [] \rightarrow [] \\ &\quad (x : xs) \rightarrow f \ x : \text{work } xs \ f \end{aligned}$$

It is interesting to note that the various users of the worker/wrapper transformation appear to realise its potential as a powerful tool for communicating a change of type through a recursive definition, but did not explore this generalisation.

The worker/wrapper terminology has also been used to describe a similar but distinct technique, namely when a wrapper of one type is exposed to an optimisation system for the purpose of applying a type conversion to a worker. Technically, this corresponds to the *adapter* design pattern (Gamma *et al.*, 1995). The rewrite system of the Glasgow Haskell Compiler (Peyton Jones *et al.*, 2001) uses this technique in its implementation of short-cut deforestation, as does the recent implementation of stream fusion (Coutts *et al.*, 2007). In both these systems, rather than wholesale inlining of a candidate function, an inlined wrapper expresses the opportunity for deforestation, but the actual computation is performed by the worker. In a real sense, these uses of workers and wrappers are part of the worker/wrapper *transformation*, which is the combination of *splitting* a function into a worker and wrapper in a correct and potentially useful manner, together with the *use* of these new functions to fulfill specific roles inside a rewrite system.

9 Conclusion and further work

In this article we presented the worker/wrapper transformation, and gave several examples of its use as an equational reasoning technique for improving the performance of functional programs. In particular, we showed how it can be used to derive more efficient programs based upon the use of accumulation, unboxed values, memoisation, and continuations. Since these are all well studied and widely used approaches to program optimisation, it is natural to ask what new value the worker/wrapper transformation brings? We see three primary benefits:

- It provides a general and systematic approach to transforming a computation of one type into an equivalent computation of another type. Such type transformations are pervasive in functional programming and reasoning, and in the efficient compilation of functional languages.
- It is straightforward to understand and apply, requiring only basic equational reasoning techniques, and often avoiding the need for induction. As such, it can readily be utilised by a wide spectrum of functional programmers as a simple but powerful new technique for refactoring their programs.
- It allows many seemingly unrelated optimisation techniques to be captured inside a single unified framework, which may help reveal new connections between existing techniques, and the discovery of new techniques.

This article is by no means the end of the worker/wrapper story, but rather opens up a number of opportunities for further research. Interesting topics for further work include mechanising the technique in an equational reasoning system such as HERA (Gill, 2006), investigating how it can be automated in an optimising compiler such as GHC (Peyton Jones *et al.*, 2001), considering programs that utilise effects (Wadler, 1992; McBride & Paterson, 2008), exploring the use of more specialised patterns of recursion (Hutton, 1999; Gibbons & Jones, 1998), and further generalising the technique using category theory (Backhouse *et al.*, 1995).

Acknowledgements

We would like to thank Roland Backhouse, Olaf Chitil, Isaac Dupree, Conal Elliott, Peter Gammie, Peter Jonsson, John Matthews, Conor McBride, Simon Peyton Jones, and Mark Tullsen for useful comments and suggestions, and Colin Runciman for the invitation to submit to JFP. This article arose from a sabbatical at Galois by the second author, for which funding from Galois is gratefully acknowledged.

Appendix

For completeness, this appendix briefly reviews the necessary semantic theory of fixed points that underlies our formalisation of the worker/wrapper transformation, and presents a proof of the rolling rule in this context.

Our formalisation is based upon the domain-theoretic approach to semantics (Schmidt, 1986), in which the basic idea is that types are *complete partial orders*

(cpo), that is, sets with a partial-ordering \sqsubseteq , a least element \perp , and limits of all non-empty chains. In turn, programs are *continuous functions*, that is, functions between cpos that preserve the partial-order and limit structure.

Now consider a recursive equation $x = f x$ that defines a value x in terms of itself and some continuous function f . A well-known fixed point theorem states that this equation has a least solution for x , denoted by $\text{fix } f$ and called the *least fixed point* of f , which is adopted as the semantics of the definition. Moreover, $\text{fix } f$ is constructed as the limit of the following infinite chain:

$$\perp \sqsubseteq f \perp \sqsubseteq f (f \perp) \sqsubseteq f (f (f \perp)) \sqsubseteq \dots$$

As a simple example of this approach, consider the equation $\text{ones} = 1 : \text{ones}$ that defines the infinite list $1 : 1 : 1 : \dots$. This definition can be rewritten as $\text{ones} = f \text{ ones}$, where f is the function defined by $f xs = 1 : xs$. Hence, the semantics of the definition is given by $\text{ones} = \text{fix } f$, and by the fixed point theorem is constructed as the limit of the infinite chain of partial lists containing increasing numbers of 1s:

$$\perp \sqsubseteq 1 : \perp \sqsubseteq 1 : 1 : \perp \sqsubseteq 1 : 1 : 1 : \perp \sqsubseteq \dots$$

The fixed point approach to recursive definitions can be realised in Haskell by defining an explicit version of the function fix as follows:

$$\begin{aligned} \text{fix} &:: (a \rightarrow a) \rightarrow a \\ \text{fix } f &= f (\text{fix } f) \end{aligned}$$

For example, evaluating the expression $\text{fix } (\lambda xs \rightarrow 1 : xs)$ using this definition gives the infinite list of ones, $1 : 1 : 1 : \dots$, as expected.

In order to prove the rolling rule, which states that $\text{fix } (g \circ f) = g (\text{fix } (f \circ g))$ for any functions f and g of the appropriate types, we exploit the following two fundamental properties of fixed points (Backhouse, 2002):

$$\begin{aligned} \text{fix } f &= f (\text{fix } f) && \text{(computation)} \\ f x \sqsubseteq x &\Rightarrow \text{fix } f \sqsubseteq x && \text{(induction)} \end{aligned}$$

The first property states that $\text{fix } f$ is a fixed point of f (an expression x satisfying $f x = x$), while the second states that $\text{fix } f$ is the least *prefix* point of f (the least expression x satisfying $f x \sqsubseteq x$). One might have expected in the latter case to state that $\text{fix } f$ is the least *fixed* point rather than just the least prefixed point, but these two notions can be shown to be equivalent, and the above formulation has the advantage of being more useful for reasoning purposes. Using these two properties, the rolling rule can now be verified by mutual inclusion:

$$\begin{aligned} &\text{fix } (g \circ f) \sqsubseteq g (\text{fix } (f \circ g)) \\ \Leftrightarrow &\quad \{ \text{induction} \} \\ &(g \circ f) (g (\text{fix } (f \circ g))) \sqsubseteq g (\text{fix } (f \circ g)) \\ \Leftrightarrow &\quad \{ \text{applying } \circ \} \\ &g (f (g (\text{fix } (f \circ g)))) \sqsubseteq g (\text{fix } (f \circ g)) \\ \Leftrightarrow &\quad \{ \text{unapplying } \circ \} \\ &g ((f \circ g) (\text{fix } (f \circ g))) \sqsubseteq g (\text{fix } (f \circ g)) \end{aligned}$$

$$\begin{aligned}
&\Leftrightarrow \{ \text{computation} \} \\
&\quad g (\text{fix } (f \circ g)) \sqsubseteq g (\text{fix } (f \circ g)) \\
&\Leftrightarrow \{ \text{reflexivity} \} \\
&\quad \text{True}
\end{aligned}$$

and

$$\begin{aligned}
&\quad g (\text{fix } (f \circ g)) \sqsubseteq \text{fix } (g \circ f) \\
&\Leftrightarrow \{ \text{computation} \} \\
&\quad g (\text{fix } (f \circ g)) \sqsubseteq (g \circ f) (\text{fix } (g \circ f)) \\
&\Leftrightarrow \{ \text{applying } \circ \} \\
&\quad g (\text{fix } (f \circ g)) \sqsubseteq g (f (\text{fix } (g \circ f))) \\
&\Leftarrow \{ g \text{ is monotonic} \} \\
&\quad \text{fix } (f \circ g) \sqsubseteq f (\text{fix } (g \circ f)) \\
&\Leftrightarrow \{ \text{above result, swapping } f \text{ and } g \} \\
&\quad \text{True}
\end{aligned}$$

We conclude this appendix by noting that the above proof of the rolling rule only relies on the basic notion of monotonic functions on partially-ordered sets, rather than the stronger notion of continuous functions on cpo's. However, working in this stronger setting automatically guarantees that the necessary fixed points always exist, by virtue of the fixed point theorem described above, which in weaker settings may become a side condition on the rolling rule.

References

- Ager, Mads Sig, Biernacki, Dariusz, Danvy, Olivier, & Midtgaard, Jan. (2003). A Functional Correspondence Between Evaluators and Abstract Machines. *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*.
- Altenkirch, Thorsten. (2001). Representations of First-order Function Types as Terminal Coalgebras. *Typed Lambda Calculi and Applications*. Lecture Notes in Computer Science, no. 2044.
- Backhouse, Roland. (2002). Galois Connections and Fixed Point Calculus. *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. LNCS Tutorial, vol. 2297. Springer.
- Backhouse, Roland, Bijsterveld, Marcel, van Geldrop, Rik, & van der Woude, Jaap. (1995). Categorical Fixed Point Calculus. *Proceedings of the 6th International Conference on Category Theory and Computer Science*. Springer-Verlag.
- Bird, Richard. (1998). *Introduction to Functional Programming using Haskell (second edition)*. Prentice Hall.
- Chitil, Olaf. (2000a). *Type-Inference Based Deforestation of Functional Programs*. Ph.D. thesis, RWTH Aachen.
- Chitil, Olaf. (2000b). Type-Inference Based Short Cut Deforestation (Nearly) Without Inlining. Chris Clack and Pieter Koopman (ed), *Proceedings of 11th International Workshop on Implementation of Functional Languages*. LNCS, no. 1868. Springer.
- Coutts, Duncan, Leshchinskiy, Roman, & Stewart, Don. (2007). Stream Fusion: From Lists to Streams to Nothing at all. *Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*. ACM Press.

- Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. (1995). *Design Patterns*. Addison-Wesley Professional.
- Gibbons, Jeremy, & Hutton, Graham. (2005). Proof Methods for Corecursive Programs. *Fundamenta Informaticae Special Issue on Program Transformation*, **66**(4).
- Gibbons, Jeremy, & Jones, Geraint. (1998). The Under-Appreciated Unfold. *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*.
- Gill, Andy. (1996). *Cheap Deforestation for Non-Strict Functional Languages*. Ph.D. thesis, University of Glasgow.
- Gill, Andy. (2006). Introducing the Haskell Equational Reasoning Assistant. *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*. ACM Press.
- Gill, Andy, Launchbury, John, & Peyton Jones, Simon. (1993). A Short Cut to Deforestation. *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. ACM Press.
- Hughes, John. (1986). A Novel Representation of Lists and its Application to the Function Reverse. *Information Processing Letters*, **22**(3).
- Hutton, Graham. (1999). A Tutorial on the Universality and Expressiveness of Fold. *Journal of Functional Programming*, **9**(4).
- Hutton, Graham, & Wright, Joel. (2004). Compiling Exceptions Correctly. *Proceedings of the 7th International Conference on Mathematics of Program Construction*. Lecture Notes in Computer Science, vol. 3125. Stirling, Scotland: Springer.
- Hutton, Graham, & Wright, Joel. (2006). Calculating an Exceptional Machine. *Trends in Functional Programming volume 5*. Intellect. Selected papers from the Fifth Symposium on Trends in Functional Programming, Munich, November 2004.
- Hutton, Graham, & Wright, Joel. (2007). What is the Meaning of These Constant Interruptions? *Journal of Functional Programming*, **17**(6).
- Launchbury, John, & Sheard, Tim. (1995). Warm Fusion: Deriving Build-Catas from Recursive Definitions. *Proceedings of the 7th ACM SIGPLAN/SIGARCH International Conference on Functional Programming Languages and Computer Architecture*. New York: ACM Press.
- McBride, Conor, & Paterson, Ross. (2008). Applicative Programming With Effects. *Journal of Functional Programming*, **18**(1).
- Meijer, Erik, Fokkinga, Maarten, & Paterson, Ross. (1991). Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. Hughes, John (ed), *Proceedings of the Conference on Functional Programming and Computer Architecture*. LNCS, no. 523. Springer-Verlag.
- Michie, Donald. (1968). Memo Functions and Machine Learning. *Nature*, **218**.
- Peyton Jones, Simon. (2003). *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press. Also available from www.haskell.org/definition.
- Peyton Jones, Simon, & Launchbury, John. (1991). Unboxed Values as First Class Citizens in a Non-strict Functional Language. *Proceedings of the Conference on Functional Programming and Computer Architecture*. Cambridge, Massachusetts: Springer-Verlag.
- Peyton Jones, Simon, & Partain, Will. (1993). Measuring the Effectiveness of a Simple Strictness Analyser. Hammond, Kevin, & O'Donnell, John (eds), *Proceedings of the 1993 Glasgow Workshop on Functional Programming*. Ayr, Scotland: Springer-Verlag.
- Peyton Jones, Simon, Tolmach, Andrew, & Hoare, Tony. (2001). Playing by the Rules: Rewriting as a Practical Optimisation Technique in GHC. *Proceedings of the 2001 ACM SIGPLAN Workshop on Haskell*. ACM Press.
- Reynolds, John C. (1972). Definitional Interpreters for Higher-Order Programming Languages. *Proceedings of the ACM Annual Conference*. ACM Press.

- Sands, David. (1998). Improvement Theory and Its Applications. Gordon, Andrew, & Pitts, Andrew (eds), *Higher Order Operational Techniques in Semantics*. Cambridge University Press.
- Santos, Andre. (1995). *Compilation by Transformation in Non-strict Functional Languages*. Ph.D. thesis, University of Glasgow.
- Schmidt, David. (1986). *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc.
- Spivey, Mike. (1990). A Functional Theory of Exceptions. *Science of Computer Programming*, **14**(1).
- Thielecke, Hayo. (2002). Comparing Control Constructs by Double-Barrelled CPS. *Higher-order and Symbolic Computation*, **15**(2/3).
- Turner, David A. (1995). Elementary Strong Functional Programming. *Proceedings of the First International Symposium on Functional Programming Languages in Education*. LNCS 1022. Springer-Verlag.
- Wadler, Philip. (1992). Monads for Functional Programming. Broy, Manfred (ed), *Proceedings of the Marktoberdorf Summer School on Program Design Calculi*. Springer-Verlag.